Optimal Growth I: The Stochastic Optimal Growth Model

Jesse Perla, Thomas J. Sargent and John Stachurski

December 4, 2020

1 Contents

- Overview 2
- The Model 3
- Computation 4
- Exercises 5
- Solutions 6

2 Overview

In this lecture we're going to study a simple optimal growth model with one agent.

The model is a version of the standard one sector infinite horizon growth model studied in

- [5], chapter 2
- [2], section 3.1
- EDTC, chapter 1
- [6], chapter 12

The technique we use to solve the model is dynamic programming.

Our treatment of dynamic programming follows on from earlier treatments in our lectures on shortest paths and job search.

We'll discuss some of the technical details of dynamic programming as we go along.

3 The Model

Consider an agent who owns an amount $y_t \in \mathbb{R}_+ := [0, \infty)$ of a consumption good at time t.

This output can either be consumed or invested.

When the good is invested it is transformed one-for-one into capital.

The resulting capital stock, denoted here by k_{t+1}, will then be used for production.

Production is stochastic, in that it also depends on a shock ξ_{t+1} realized at the end of the current period.

Next period output is

$$y_{t+1} := f(k_{t+1})\xi_{t+1}$$

where $f \colon \mathbb{R}_+ \to \mathbb{R}_+$ is called the production function.

The resource constraint is

$$k_{t+1} + c_t \leq y_t \tag{1}$$

and all variables are required to be nonnegative.

3.1 Assumptions and Comments

In what follows,

- The sequence $\{\xi_t\}$ is assumed to be IID.
- The common distribution of each ξ_t will be denoted ϕ.
- The production function f is assumed to be increasing and continuous.
- Depreciation of capital is not made explicit but can be incorporated into the production function.

While many other treatments of the stochastic growth model use k_t as the state variable, we will use y_t.

This will allow us to treat a stochastic model while maintaining only one state variable.

We consider alternative states and timing specifications in some of our other lectures.

3.2 Optimization

Taking y_0 as given, the agent wishes to maximize

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(c_t)\right] \tag{2}$$

subject to

$$y_{t+1} = f(y_t - c_t)\xi_{t+1} \quad \text{and} \quad 0 \leq c_t \leq y_t \quad \text{for all } t \tag{3}$$

where

- u is a bounded, continuous and strictly increasing utility function and
- $\beta \in (0, 1)$ is a discount factor

In (3) we are assuming that the resource constraint (1) holds with equality — which is reasonable because u is strictly increasing and no output will be wasted at the optimum.

In summary, the agent's aim is to select a path $c_0, c_1, c_2, \ldots$ for consumption that is

1. nonnegative,

2. feasible in the sense of (1),
3. optimal, in the sense that it maximizes (2) relative to all other feasible consumption sequences, and
4. *adapted*, in the sense that the action c_t depends only on observable outcomes, not future outcomes such as ξ_{t+1}

In the present context

- y_t is called the *state* variable — it summarizes the "state of the world" at the start of each period.
- c_t is called the *control* variable — a value chosen by the agent each period after observing the state.

3.3 The Policy Function Approach

One way to think about solving this problem is to look for the best **policy function**.

A policy function is a map from past and present observables into current action.

We'll be particularly interested in **Markov policies**, which are maps from the current state y_t into a current action c_t.

For dynamic programming problems such as this one (in fact for any Markov decision process), the optimal policy is always a Markov policy.

In other words, the current state y_t provides a sufficient statistic for the history in terms of making an optimal decision today.

This is quite intuitive but if you wish you can find proofs in texts such as [5] (section 4.1).

Hereafter we focus on finding the best Markov policy.

In our context, a Markov policy is a function $\sigma : \mathbb{R}_+ \to \mathbb{R}_+$, with the understanding that states are mapped to actions via

$$c_t = \sigma(y_t) \quad \text{for all } t$$

In what follows, we will call σ a *feasible consumption policy* if it satisfies

$$0 \leq \sigma(y) \leq y \quad \text{for all} \quad y \in \mathbb{R}_+ \tag{4}$$

In other words, a feasible consumption policy is a Markov policy that respects the resource constraint.

The set of all feasible consumption policies will be denoted by Σ.

Each $\sigma \in \Sigma$ determines a continuous state Markov process $\{y_t\}$ for output via

$$y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}, \quad y_0 \text{ given} \tag{5}$$

This is the time path for output when we choose and stick with the policy σ.

We insert this process into the objective function to get

$$\mathbb{E}\left[\sum_{t=0}^{\infty}\beta^t u(c_t)\right] = \mathbb{E}\left[\sum_{t=0}^{\infty}\beta^t u(\sigma(y_t))\right] \tag{6}$$

This is the total expected present value of following policy σ forever, given initial income y_0.

The aim is to select a policy that makes this number as large as possible.

The next section covers these ideas more formally.

3.4 Optimality

The **policy value function** v_σ associated with a given policy σ is the mapping defined by

$$v_\sigma(y) = \mathbb{E}\left[\sum_{t=0}^{\infty}\beta^t u(\sigma(y_t))\right] \tag{7}$$

when $\{y_t\}$ is given by (5) with $y_0 = y$.

In other words, it is the lifetime value of following policy σ starting at initial condition y.

The **value function** is then defined as

$$v^*(y) := \sup_{\sigma\in\Sigma} v_\sigma(y) \tag{8}$$

The value function gives the maximal value that can be obtained from state y, after considering all feasible policies.

A policy $\sigma \in \Sigma$ is called **optimal** if it attains the supremum in (8) for all $y \in \mathbb{R}_+$.

3.5 The Bellman Equation

With our assumptions on utility and production function, the value function as defined in (8) also satisfies a **Bellman equation**.

For this problem, the Bellman equation takes the form

$$w(y) = \max_{0\le c\le y}\left\{u(c) + \beta\int w(f(y-c)z)\phi(dz)\right\} \qquad (y \in \mathbb{R}_+) \tag{9}$$

This is a *functional equation in* w.

The term $\int w(f(y-c)z)\phi(dz)$ can be understood as the expected next period value when

- w is used to measure value
- the state is y
- consumption is set to c

As shown in EDTC, theorem 10.1.11 and a range of other texts.

The value function v^* satisfies the Bellman equation

In other words, (9) holds when $w = v^*$.

The intuition is that maximal value from a given state can be obtained by optimally trading off

- current reward from a given action, vs
- expected discounted future value of the state resulting from that action

The Bellman equation is important because it gives us more information about the value function.

It also suggests a way of computing the value function, which we discuss below.

3.6 Greedy policies

The primary importance of the value function is that we can use it to compute optimal policies.

The details are as follows.

Given a continuous function w on $\mathbb{R}_+$, we say that $\sigma \in \Sigma$ is w-**greedy** if $\sigma(y)$ is a solution to

$$\max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y-c)z)\phi(dz) \right\} \tag{10}$$

for every $y \in \mathbb{R}_+$.

In other words, $\sigma \in \Sigma$ is w-greedy if it optimally trades off current and future rewards when w is taken to be the value function.

In our setting, we have the following key result

> *A feasible consumption policy is optimal if and only if it is v^*-greedy*

The intuition is similar to the intuition for the Bellman equation, which was provided after (9).

See, for example, theorem 10.1.11 of EDTC.

Hence, once we have a good approximation to v^*, we can compute the (approximately) optimal policy by computing the corresponding greedy policy.

The advantage is that we are now solving a much lower dimensional optimization problem.

3.7 The Bellman Operator

How, then, should we compute the value function?

One way is to use the so-called **Bellman operator**.

(An operator is a map that sends functions into functions)

The Bellman operator is denoted by T and defined by

$$Tw(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y-c)z)\phi(dz) \right\} \qquad (y \in \mathbb{R}_+) \tag{11}$$

In other words, T sends the function w into the new function Tw defined (11).

By construction, the set of solutions to the Bellman equation (9) *exactly coincides with* the set of fixed points of T.

For example, if $Tw = w$, then, for any $y \geq 0$,

$$w(y) = Tw(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\}$$

which says precisely that w is a solution to the Bellman equation.

It follows that v^* is a fixed point of T.

3.8 Review of Theoretical Results

One can also show that T is a contraction mapping on the set of continuous bounded functions on $\mathbb{R}_+$ under the supremum distance

$$\rho(g, h) = \sup_{y \geq 0} |g(y) - h(y)|$$

See EDTC, lemma 10.1.18.

Hence it has exactly one fixed point in this set, which we know is equal to the value function.

It follows that

- The value function v^* is bounded and continuous.
- Starting from any bounded and continuous w, the sequence $w, Tw, T^2w, \ldots$ generated by iteratively applying T converges uniformly to v^*.

This iterative method is called **value function iteration**.

We also know that a feasible policy is optimal if and only if it is v^*-greedy.

It's not too hard to show that a v^*-greedy policy exists (see EDTC, theorem 10.1.11 if you get stuck).

Hence at least one optimal policy exists.

Our problem now is how to compute it.

3.9 Unbounded Utility

The results stated above assume that the utility function is bounded.

In practice economists often work with unbounded utility functions — and so will we.

In the unbounded setting, various optimality theories exist.

Unfortunately, they tend to be case specific, as opposed to valid for a large range of applications.

Nevertheless, their main conclusions are usually in line with those stated for the bounded case just above (as long as we drop the word "bounded").

Consult, for example, section 12.2 of EDTC, [1] or [3].

4 Computation

Let's now look at computing the value function and the optimal policy.

4.1 Fitted Value Iteration

The first step is to compute the value function by value function iteration.

In theory, the algorithm is as follows

1. Begin with a function w — an initial condition.
2. Solving (11), obtain the function Tw.
3. Unless some stopping condition is satisfied, set $w = Tw$ and go to step 2.

This generates the sequence $w, Tw, T^2w, ...$

However, there is a problem we must confront before we implement this procedure: The iterates can neither be calculated exactly nor stored on a computer.

To see the issue, consider (11).

Even if w is a known function, unless Tw can be shown to have some special structure, the only way to store it is to record the value $Tw(y)$ for every $y \in \mathbb{R}_+$.

Clearly this is impossible.

What we will do instead is use **fitted value function iteration**.

The procedure is to record the value of the function Tw at only finitely many "grid" points $y_1 < y_2 < \cdots < y_I$ and reconstruct it from this information when required.

More precisely, the algorithm will be

1. Begin with an array of values $\{w_1, \ldots, w_I\}$ representing the values of some initial function w on the grid points $\{y_1, \ldots, y_I\}$.
2. Build a function $\hat{w}$ on the state space $\mathbb{R}_+$ by interpolation or approximation, based on these data points.
3. Obtain and record the value $T\hat{w}(y_i)$ on each grid point y_i by repeatedly solving (11).
4. Unless some stopping condition is satisfied, set $\{w_1, \ldots, w_I\} = \{T\hat{w}(y_1), \ldots, T\hat{w}(y_I)\}$ and go to step 2.

How should we go about step 2?

This is a problem of function approximation, and there are many ways to approach it.

What's important here is that the function approximation scheme must not only produce a good approximation to Tw, but also combine well with the broader iteration algorithm described above.

One good choice from both respects is continuous piecewise linear interpolation (see this paper for further discussion).

The next figure illustrates piecewise linear interpolation of an arbitrary function on grid points $0, 0.2, 0.4, 0.6, 0.8, 1$.

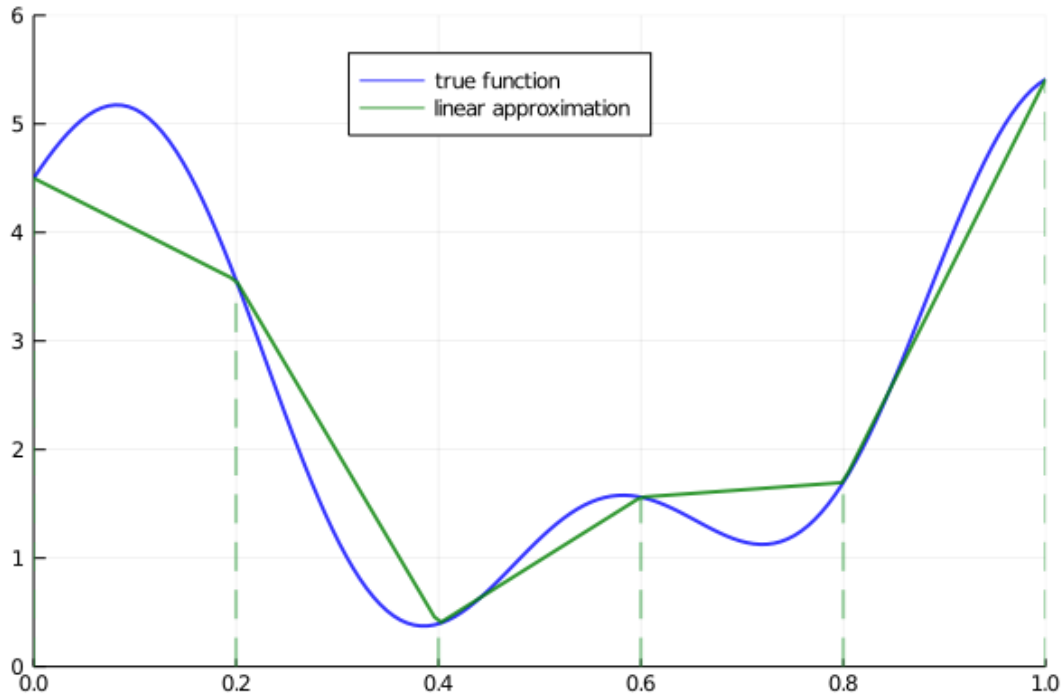
4.2 Setup

```
In [1]: using InstantiateFromURL
        # optionally add arguments to force installation: instantiate = true,
↪precompile = true
        github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
```

```
In [2]: using LinearAlgebra, Statistics
        using Plots, QuantEcon, Interpolations, NLsolve, Optim, Random
        gr(fmt = :png);
```

```
In [3]: f(x) = 2 .* cos.(6x) .+ sin.(14x) .+ 2.5
        c_grid = 0:.2:1
        f_grid = range(0,  1, length = 150)

        Af = LinearInterpolation(c_grid, f(c_grid))

        plt = plot(xlim = (0,1), ylim = (0,6))
        plot!(plt, f, f_grid, color = :blue, lw = 2, alpha = 0.8, label = "true
↪function")
        plot!(plt, f_grid, Af.(f_grid), color = :green, lw = 2, alpha = 0.8,
              label = "linear approximation")
        plot!(plt, f, c_grid, seriestype = :sticks, linestyle = :dash, linewidth =
↪2, alpha =
        0.5,
              label = "")
        plot!(plt, legend = :top)
```

Out[3]:



Another advantage of piecewise linear interpolation is that it preserves useful shape properties such as monotonicity and concavity / convexity.

4.3 The Bellman Operator

Here's a function that implements the Bellman operator using linear interpolation

```
In [4]: using Optim

        function T(w, grid, β, u, f, shocks; compute_policy = false)
            w_func = LinearInterpolation(grid, w)
            # objective for each grid point
            objectives = (c -> u(c) + β * mean(w_func.(f(y - c) .* shocks)) for y
 ↪in grid)
            results = maximize.(objectives, 1e-10, grid) # solver result for each
 ↪grid point

            Tw = Optim.maximum.(results)
            if compute_policy
                σ = Optim.maximizer.(results)
                return Tw, σ
            end

            return Tw
        end
```

```
Out[4]: T (generic function with 1 method)
```

Notice that the expectation in (11) is computed via Monte Carlo, using the approximation

$$\int w(f(y-c)z)\phi(dz) \approx \frac{1}{n}\sum_{i=1}^{n} w(f(y-c)\xi_i)$$

where $\{\xi_i\}_{i=1}^n$ are IID draws from ϕ.

Monte Carlo is not always the most efficient way to compute integrals numerically but it does have some theoretical advantages in the present setting.

(For example, it preserves the contraction mapping property of the Bellman operator — see, e.g., [4])

4.4 An Example

Let's test out our operator when

- $f(k) = k^\alpha$
- $u(c) = \ln c$
- ϕ is the distribution of $\exp(\mu + \sigma\zeta)$ when ζ is standard normal

As is well-known (see [2], section 3.1.2), for this particular problem an exact analytical solution is available, with

$$v^*(y) = \frac{\ln(1-\alpha\beta)}{1-\beta} + \frac{(\mu + \alpha\ln(\alpha\beta))}{1-\alpha}\left[\frac{1}{1-\beta} - \frac{1}{1-\alpha\beta}\right] + \frac{1}{1-\alpha\beta}\ln y \tag{12}$$

The optimal consumption policy is

$$\sigma^*(y) = (1 - \alpha\beta)y$$

Let's code this up now so we can test against it below

In [5]:
```
α = 0.4
β = 0.96
μ = 0
s = 0.1

c1 = log(1 - α * β) / (1 - β)
c2 = (μ + α * log(α * β)) / (1 - α)
c3 = 1 / (1 - β)
c4 = 1 / (1 - α * β)

# Utility
u(c) = log(c)

∂u∂c(c) = 1 / c

# Deterministic part of production function
f(k) = k^α

f'(k) = α * k^(α - 1)

# True optimal policy
c_star(y) = (1 - α * β) * y

# True value function
v_star(y) = c1 + c2 * (c3 - c4) + c4 * log(y)
```

Out[5]:
```
v_star (generic function with 1 method)
```

4.5 A First Test

To test our code, we want to see if we can replicate the analytical solution numerically, using fitted value function iteration.

We need a grid and some shock draws for Monte Carlo integration.

In [6]:
```
using Random
Random.seed!(42) # For reproducible results.

grid_max = 4         # Largest grid point
grid_size = 200      # Number of grid points
shock_size = 250     # Number of shock draws in Monte Carlo integral

grid_y = range(1e-5,  grid_max, length = grid_size)
shocks = exp.(μ .+ s * randn(shock_size))
```

Now let's do some tests.

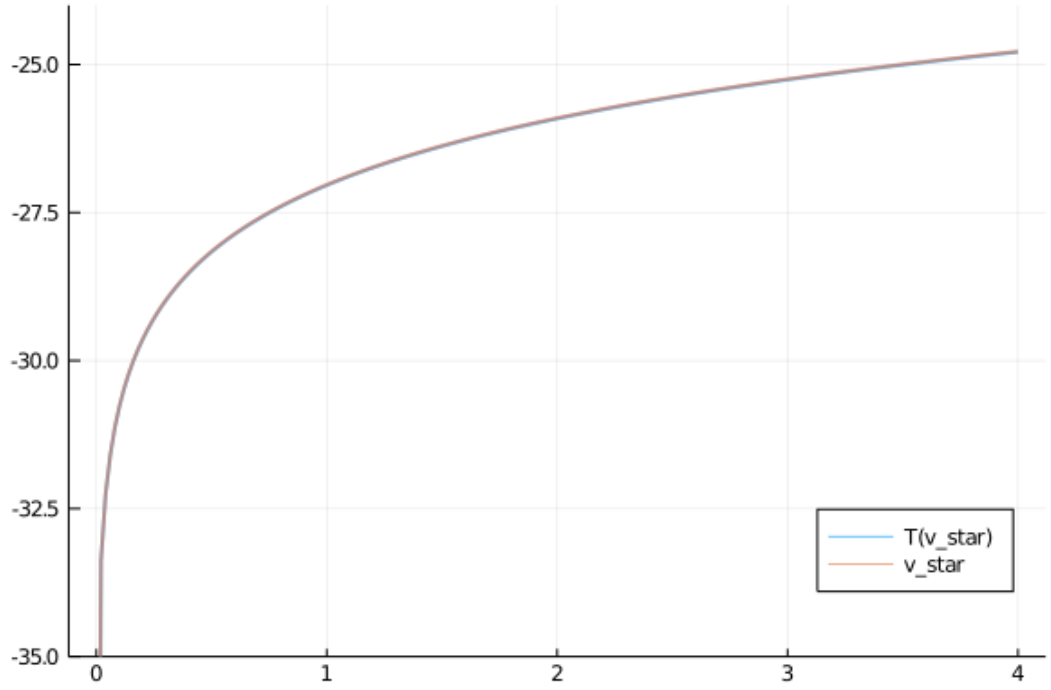
As one preliminary test, let's see what happens when we apply our Bellman operator to the exact solution v^*.

In theory, the resulting function should again be v^*.

In practice we expect some small numerical error.

In [7]:
```
w = T(v_star.(grid_y), grid_y, β, log, k -> k^α, shocks)

plt = plot(ylim = (-35,-24))
plot!(plt, grid_y, w, linewidth = 2, alpha = 0.6, label = "T(v_star)")
plot!(plt, v_star, grid_y, linewidth = 2, alpha=0.6, label = "v_star")
plot!(plt, legend = :bottomright)
```

Out[7]:



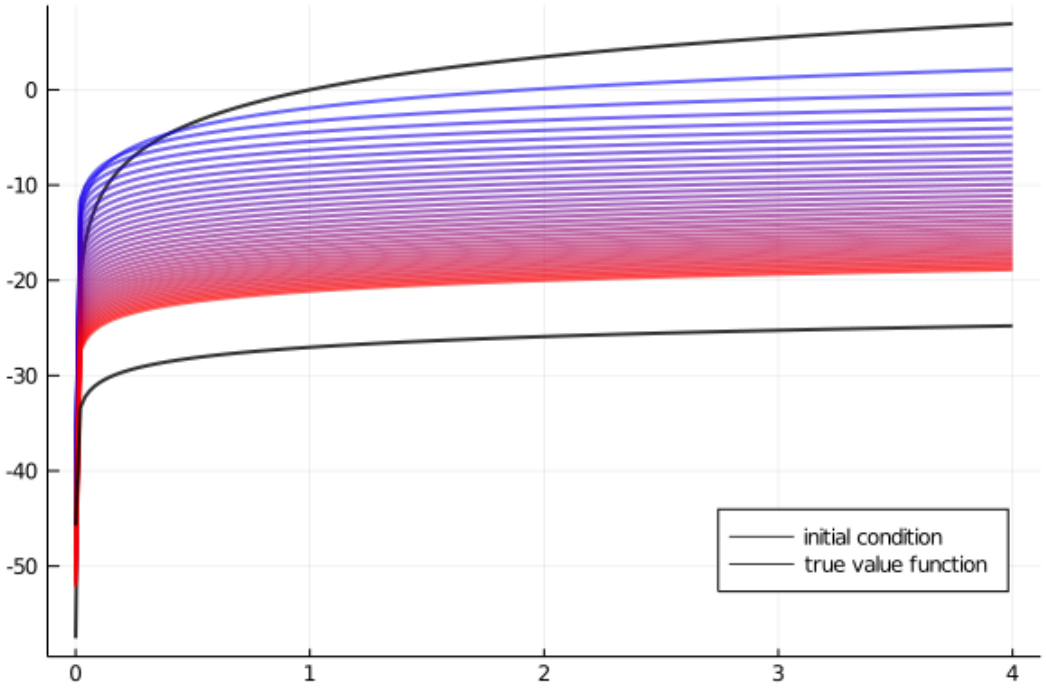
The two functions are essentially indistinguishable, so we are off to a good start.

Now let's have a look at iterating with the Bellman operator, starting off from an arbitrary initial condition.

The initial condition we'll start with is $w(y) = 5\ln(y)$

In [8]:
```
w = 5 * log.(grid_y)  # An initial condition -- fairly arbitrary
n = 35

plot(xlim = (extrema(grid_y)), ylim = (-50, 10))
lb = "initial condition"
plt = plot(grid_y, w, color = :black, linewidth = 2, alpha = 0.8, label = lb)
for i in 1:n
    w = T(w, grid_y, β, log, k -> k^α, shocks)
    plot!(grid_y, w, color = RGBA(i/n, 0, 1 - i/n, 0.8), linewidth = 2,
↪alpha = 0.6,
          label = "")
end

lb = "true value function"
plot!(plt, v_star, grid_y, color = :black, linewidth = 2, alpha = 0.8,
↪label = lb)
plot!(plt, legend = :bottomright)
```

Out[8]:



The figure shows

1. the first 36 functions generated by the fitted value function iteration algorithm, with hotter colors given to higher iterates
2. the true value function v^* drawn in black

The sequence of iterates converges towards v^*.

We are clearly getting closer.

We can write a function that computes the exact fixed point

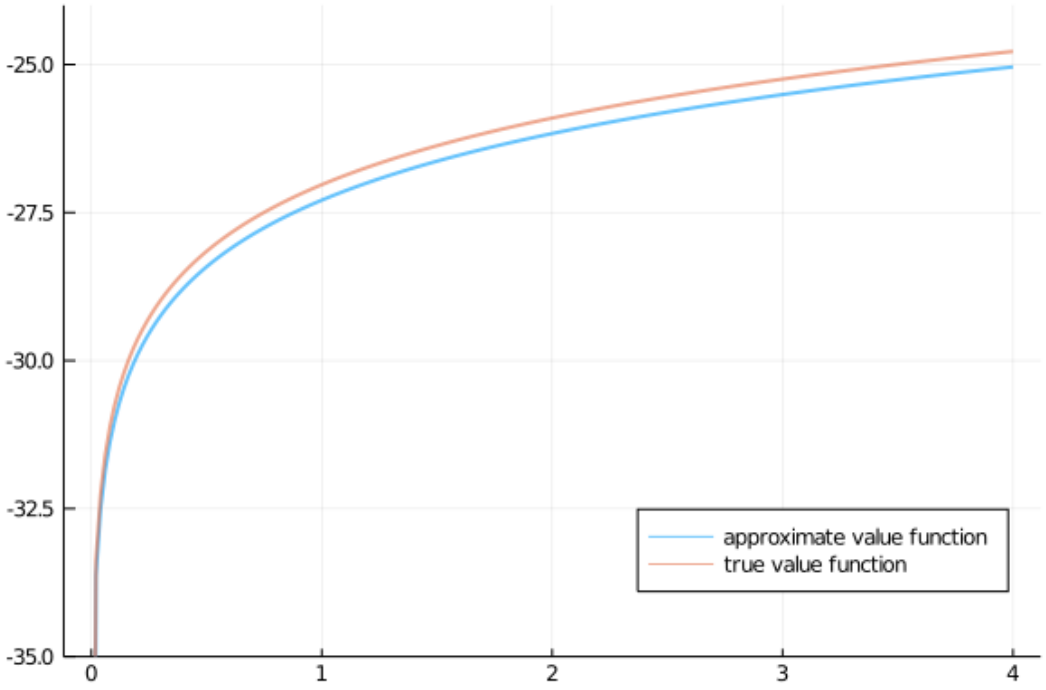
```
In [9]: function solve_optgrowth(initial_w; tol = 1e-6, max_iter = 500)
            fixedpoint(w -> T(w, grid_y, β, u, f, shocks), initial_w).zero # gets
 ↪returned
        end
```

```
Out[9]: solve_optgrowth (generic function with 1 method)
```

We can check our result by plotting it against the true value

```
In [10]: initial_w = 5 * log.(grid_y)
         v_star_approx = solve_optgrowth(initial_w)

         plt = plot(ylim = (-35, -24))
         plot!(plt, grid_y, v_star_approx, linewidth = 2, alpha = 0.6,
               label = "approximate value function")
         plot!(plt, v_star, grid_y, linewidth = 2, alpha = 0.6, label = "true
 ↪value function")
         plot!(plt, legend = :bottomright)
```

Out[10]:



The figure shows that we are pretty much on the money.

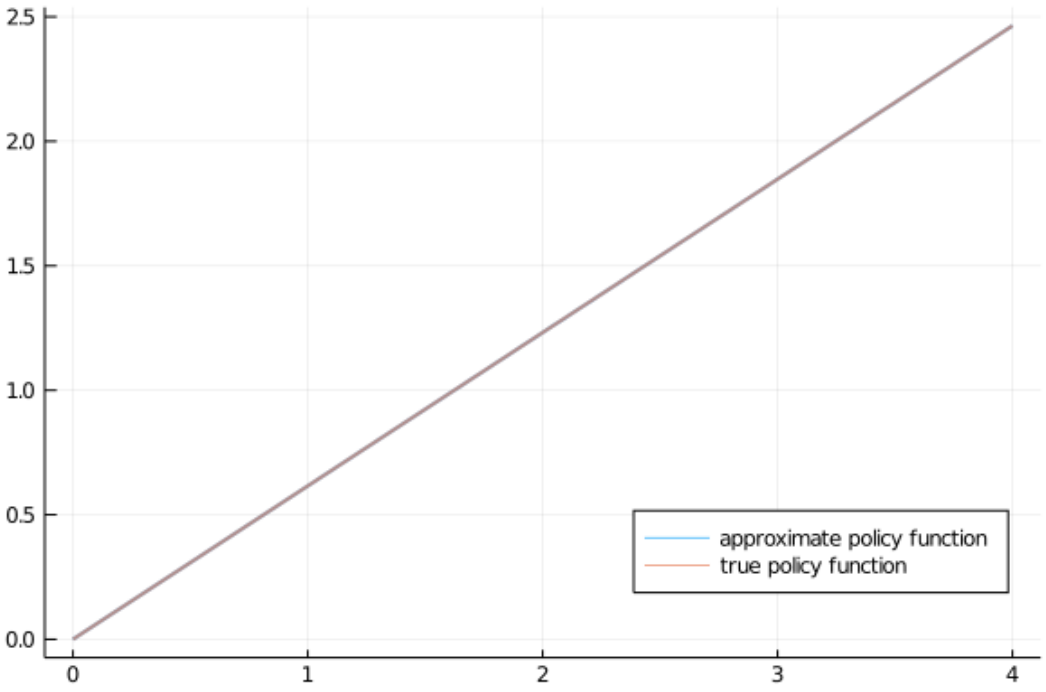
4.6 The Policy Function

To compute an approximate optimal policy, we take the approximate value function we just calculated and then compute the corresponding greedy policy.

The next figure compares the result to the exact solution, which, as mentioned above, is $\sigma(y) = (1 - \alpha\beta)y$.

In [11]:
```
Tw, σ = T(v_star_approx, grid_y, β, log, k -> k^α, shocks;
                                 compute_policy = true)
        cstar = (1 - α * β) * grid_y

        plt = plot(grid_y, σ, lw=2, alpha=0.6, label = "approximate policy␣
↪function")
        plot!(plt, grid_y, cstar, lw = 2, alpha = 0.6, label = "true policy␣
↪function")
        plot!(plt, legend = :bottomright)
```

Out[11]:



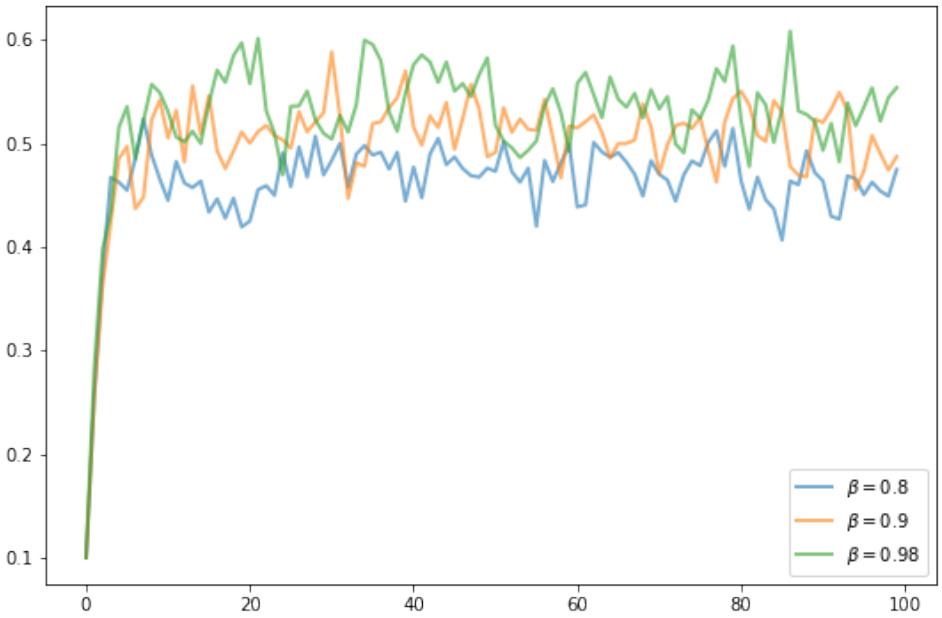
The figure shows that we've done a good job in this instance of approximating the true policy.

5 Exercises

5.1 Exercise 1

Once an optimal consumption policy σ is given, income follows (5).

The next figure shows a simulation of 100 elements of this sequence for three different discount factors (and hence three different policies).



In each sequence, the initial condition is $y_0 = 0.1$.

The discount factors are `discount_factors = (0.8, 0.9, 0.98)`.

We have also dialed down the shocks a bit.

```
In [12]: s = 0.05
         shocks = exp.(μ .+ s * randn(shock_size))
```

Otherwise, the parameters and primitives are the same as the log linear model discussed earlier in the lecture.

Notice that more patient agents typically have higher wealth.

Replicate the figure modulo randomness.

6 Solutions

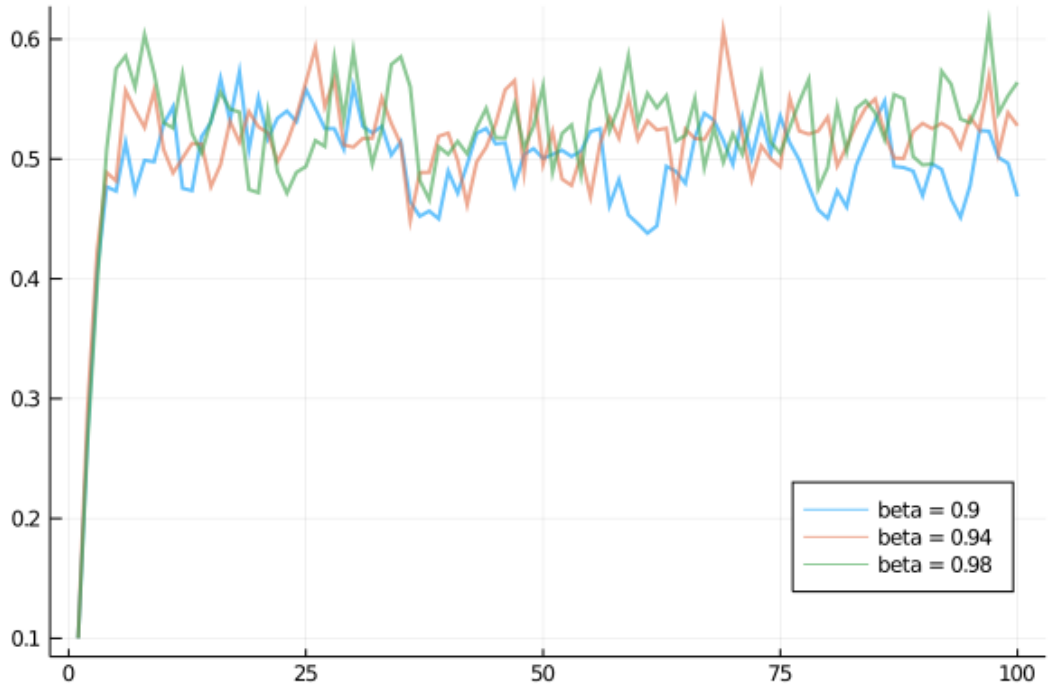
6.1 Exercise 1

Here's one solution (assuming as usual that you've executed everything above)

```
In [13]: function simulate_og(σ, y0 = 0.1, ts_length=100)
             y = zeros(ts_length)
             ξ = randn(ts_length-1)
             y[1] = y0
             for t in 1:(ts_length-1)
                 y[t+1] = (y[t] - σ(y[t]))^α * exp(μ + s * ξ[t])
             end
             return y
         end
```

```
plt = plot()

for β in (0.9, 0.94, 0.98)
    Tw = similar(grid_y)
    initial_w = 5 * log.(grid_y)

    v_star_approx = fixedpoint(w -> T(w, grid_y, β, u, f, shocks),
                               initial_w).zero
    Tw, σ = T(v_star_approx, grid_y, β, log, k -> k^α, shocks,
                             compute_policy = true)
    σ_func = LinearInterpolation(grid_y, σ)
    y = simulate_og(σ_func)

    plot!(plt, y, lw = 2, alpha = 0.6, label = label = "beta = $β")
end

plot!(plt, legend = :bottomright)
```

Out[13]:



References

[1] Takashi Kamihigashi. Elementary results on solutions to the bellman equation of dynamic programming: existence, uniqueness, and convergence. Technical report, Kobe University, 2012.

[2] L Ljungqvist and T J Sargent. *Recursive Macroeconomic Theory*. MIT Press, 4 edition, 2018.

[3] V Filipe Martins-da Rocha and Yiannis Vailakis. Existence and Uniqueness of a Fixed Point for Local Contractions. *Econometrica*, 78(3):1127–1141, 2010.

[4] Jenő Pál and John Stachurski. Fitted value function iteration with probability one contractions. *Journal of Economic Dynamics and Control*, 37(1):251–264, 2013.

[5] N L Stokey, R E Lucas, and E C Prescott. *Recursive Methods in Economic Dynamics*. Harvard University Press, 1989.

[6] R K Sundaram. *A First Course in Optimization Theory*. Cambridge University Press, 1996.